Generative artificial intelligence for model-based graphical programming in automotive function development

Abdelrahman Abdalla[a], Harsh Pandey[a,b], Behzad Shomali[a], Joschka Schaub[b], Arne Müller[b], Markus Eisenbarth[a], Jakob Andert[a,*]

[a]*Teaching and Research Area Mechatronics in Mobile Propulsion, RWTH Aachen University, Forckenbeckstraße 4, Aachen, 52074, Germany*
[b]*FEV Europe GmbH, Neuenhofstraße 181, Aachen, 52078, Germany*

Abstract

The recent development in large language models (LLMs) enhanced their ability to write code in different programming languages based on natural language text input. However, to the best of the authors' knowledge, there has been no LLM-based study that focuses on the generation of functions in a graphic programming language. Developing a specialised LLM for generating functions in *MATLAB Simulink*, a widely utilised graphical programming environment in the automotive industry, holds promise for significantly boosting efficiency and reducing development time in function development. The primary objective of this study is to automate: (1) the generation of *MATLAB Simulink* functions from software requirements, and (2) the creation of documentation for existing functions. To achieve this, two open-source foundational LLMs with 7 billion parameter (7B) have been fine-tuned. To overcome the scarcity of training data, using the chat generative pre-trained transformer (ChatGPT) *GPT-3.5-turbo* language model, a modified self generated instructions (*self-instruct*) approach is employed to generate synthetic training data. Subsequently, as a post-processing step, a repair algorithm is utilised to improve the quality of the generated data. This study reports on the quality of the generated data and the improvements made to it. Furthermore, a testing benchmark is developed to test LLMs performance for graphical programming in *MATLAB Simulink*. We show that the fine-tuned LLMs, despite having 7B parameters, outperformed ChatGPT *GPT-3.5-turbo* by 53% and ChatGPT more advanced *GPT-4* language model by 20% on the benchmark.

Keywords: Large Language Models, Function Development, Fine-Tuning, Graphical Programming, Synthetic Data

1. Introduction

Recent advancements in Large Language Models (LLMs) [1, 2, 3] have garnered considerable attention owing to their diverse capabilities. Trained on extensive datasets and containing billions of parameters, LLMs such as OpenAI's ChatGPT have demonstrated impressive capabilities across a diverse range of natural language tasks [1, 2]. This trend has extended to open-source LLMs [4, 5, 6, 7, 8], with Meta introducing Llama [5] and Llama 2 [6] - a collection of foundation models ranging from 7B to 70B parameters, and Mistral AI developing the Mistral 7B language model that has demonstrated a remarkable performance across benchmarks encompassing reasoning, comprehension, and code generation [4]. These large language models, while exhibiting various emergent qualities [1, 2], can also be instruction-tuned (or fine-tuned) to align their outputs with human preferences - enabling the models to adapt to specific domains or learn specialised knowledge without requiring significant resources [3]. Notable examples of fine-tuned LLMs are ChatGPT, tuned to act as a conversation agent [3], or Code-Llama - a fine-tuned Llama model proficient in code generation [9]. A fine-tuned LLM, through the generation of code snippets from natural language [10], can help in reducing the time required for function and software development [11, 12, 13]. To substantiate this assertion, in [14], developers were divided into two teams, each tasked with implementing an HTTP server in

*Corresponding author
Email address: andert@mmp.rwth-aachen.de (Jakob Andert)

JavaScript. The treated group with access to GitHub Copilot, an AI pair programmer powered by OpenAI's generative AI model [10], was able to complete the task 55.8% faster than the control group [14].

Signal based graphical programming environments, such as MATLAB Simulink and TargetLink, are widely used in automotive simulation and control algorithm development [15], in contrast to text-based programming languages. These environments pose unique challenges due to the complex block connections required to execute specific sequences of operations [16]. Current LLMs face significant limitations in this area, as demonstrated by a preliminary evaluation of ChatGPT, Mistral 7B, Llama, and Code-Llama. This assessment highlights the sub-optimal performance of these LLMs in generating functions within the MATLAB Simulink environment. These shortcomings can stem from the fundamental differences between textual and graphical programming paradigms [16] and the limited amount of open-access training data available for this specialised task [17, 18, 19].

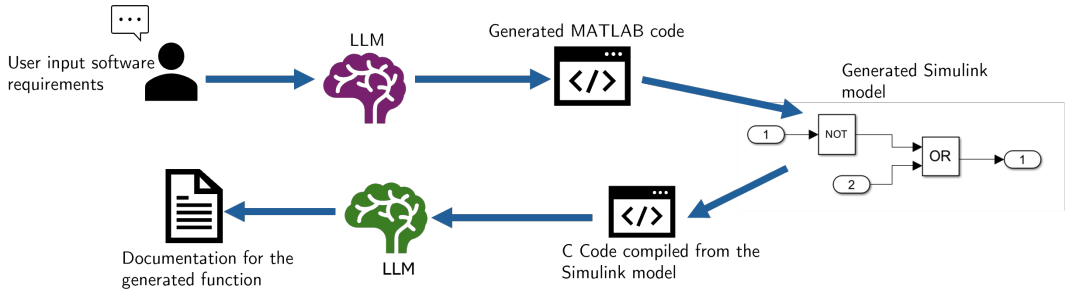


Figure 1: Proposed workflow for using LLMs to generate functions in MATLAB Simulink and document generated functions.

As LLMs require substantial training data to specialise in particular tasks [20, 3], the generation of this amount of data represents a critical challenge in training the LLMs for a specific task [21]. Regardless of the target task, the creation of correct and relevant instructions is an arduous task for humans, requiring creativity, diversity, and expertise. The Self-Instruct framework [20] addresses the scarcity of human-written instructions, which is important for aligning LLMs with specific tasks by bootstrapping LLMs using their own generations. The framework used 175 human-written statements and GPT-3-Vanilla to generate 52,000 statements paired with 82,000 input-output instances. Additionally, a filtering process was implemented to identify and remove invalid and similar statements. Subsequently, the Evol-Instruct framework [22] was proposed to overcome the limitations in generating complex instructions for LLM training by employing LLMs to generate diverse and more complex instruction data through in-depth and in-breadth evolving strategies. The same 175 human-written instructions were used to generate 250,000 instructions with input-output instances [22]. The Evol-Instruct approach was further adapted to code generation, and the generated data was used to fine-tune a coding LLM named WizzardCoder [7] that exhibited remarkable performance over prominent code generation benchmarks. However, instruct approaches' biggest limitation lies in their reliance on LLMs for data generation, resulting in the inheritance of all the limitations associated with the specific LLM used for this purpose [20].

The automotive industry is increasingly adopting model-based approaches for function development (FD) [23, 24], where functions are developed in graphical programming environments, based on software requirements and then documented [25]. Although the benefits of adopting model-based development are well documented, it can be an effort-intensive activity [26]. Given the advancements in LLMs and their demonstrated ability in coding tasks [14, 10], it is pertinent to explore the potential extension of these advancements to graphical programming function development practices, particularly in specialised domains such as automotive engineering. In light of this, this research seeks to investigate the efficacy of LLMs in generating functions directly from software requirements. We aim to streamline the function development process, ultimately reducing the time required for development while maintaining or even improving the quality of the produced functions.

In MATLAB Simulink, functions are typically saved in the *.slx* format, which is not directly readable by LLMs. Therefore, in this study, rather than working directly with *.slx* files, we are evaluating the model's capability to generate MATLAB code that creates the corresponding Simulink function for a specific requirement. We, further, compile the functions into C code for documentation and then test the LLM on documenting the compiled code. This process is

illustrated in Figure 1.

To conquer data scarcity, we adapt the Self-Instruct and Evol-Instruct frameworks to devise a novel Sim-Instruct framework tailored specifically for MATLAB Simulink code generation. Our framework uses a similar approach to Self-Instruct for the data generation process, supplemented by prompts adapted from Evol-Instruct to increase the reasoning and complexity of generated instructions. In order to overcome the inheritance limitations of instruct approaches in our framework, we develop a repair algorithm that autonomously executes the codes generated by Sim-Instruct, catches the implementation errors, and solves a large percentage of syntactical bugs present in the generated code.

This study is centered on the investigation of fine-tuning open-source LLMs for the specific use case of function development, with a focus on generating functions within the graphical programming environment MATLAB Simulink based on software requirements, as well as documenting existing functions. Through this research, we aim to utilise LLMs in streamlining function development processes within the automotive industry.

The paper is structured as follows: Section 2 provides an in-depth exploration of the Sim-Instruct framework, encompassing detailed discussions on training data preparation for both graphical programming and documentation, and a discussion of hyper-parameter tuning. Additionally, an evaluation scheme for the generated training data is presented and the developed repair algorithm and instruction fine-tuning are discussed. In Section 3, the diversity and quality of the data generated by Sim-Instruct will be discussed, along with the presentation of the benchmark developed to evaluate various LLMs in graphical programming. Following this, the performance of the fine-tuned models will be compared to the state-of-the-art on the created benchmark. Finally, Section 4 presents the conclusions drawn from the study's outcomes, along with limitations and avenues for future research.

2. Materials and Methods

2.1. Sim-Instruct Framework

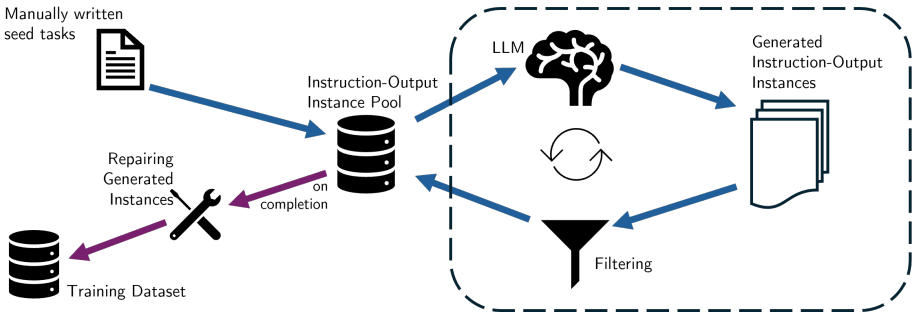


Figure 2: A high-level overview of Sim-Instruct Framework.

The Sim-Instruct framework, shown in Figure 2, utilises a slightly modified Self-Instruct methodology and leverages a singular LM (GPT-3.5-turbo) to generate both the instruction and output to ease inference. The process operates in a similar bootstrapping fashion, using a small set of human-written instruction-output pairs (henceforth referred to as seed tasks). In every step, instruction-output pairs are sampled from this pool as in-context examples augmented by a prompting scheme inspired by Evol-Instruct and containing further task-specific instructions. The generated instruction-output pairs are filtered to remove unfinished outputs and maintain diversity, and are finally added to the initial seed-task pool. After the completion of this data generation, syntactical errors are identified, logged, and repaired where possible, followed by removing any irreparable erroneous data. The prompting scheme utilised is shown below:

> The output should be a MATLAB code that generates the Simulink model for the corresponding instruction.
>
> The type of instructions should be a description of a functional requirement.
>
> Increase the reasoning of initial instruction.

Generate more deep instruction.

Add constraints to the initial instruction.

Add more reasoning steps to the initial instruction.

Complicate the initial instruction.

Generate completely new instructions.

2.2. Generated Data Evaluation Scheme

Accounting for the difficulty in implementing an automatic evaluation and capturing the accuracy and efficacy of the LLM-generated outputs comprehensively, a 4-level rating system was constructed in conjunction with experts of FEV Europe GmbH, one of the world's largest independent development service providers for the automotive industry. The generated outputs were then manually evaluated by the authors of this paper. The outputs were categorised into the following four categories:

- **Grade A:** The generated output is correct.
- **Grade B:** The generated output is largely correct but requires minor corrections. This can be represented by a missing line connection or mislabelled parameters in block declarations for function development. For documentation generation, this can be represented by inaccurate translation of signal types in the variable descriptions.
- **Grade C:** The generated output is relevant but contains significant errors. For function development, this can be represented by missing operators or blocks. For documentation generation, this can be exemplified by misrepresenting variable types or incorrectly translating variable names.
- **Grade D:** The generated output is invalid and unusable.

2.3. Training Data Preparation

Function Development Requirements specification is central in the development process of automotive software. All subsequent development, testing, and refinement are based on the defined requirements. Although the most common way to communicate and document the system requirements is the usage of natural language [27], this practice is susceptible to several problems [28]. To address the possible ambiguity, vagueness, complexity, and untestability that can arise in using natural language requirements, multiple studies have proposed the adoption of requirement templates. These templates define the syntactical structure of a written requirement to enable clarity in its subsequent communication and interpretation. Considering these factors and to maximise the amount of information communicated through a single requirement, a structure similar to NASA's proposed structural natural language [29] - FRETish - is used to elicit the seed tasks' requirements. The following is a list of the Simulink libraries and the corresponding blocks used in the created seed tasks in this study:

- **Math Operations:** Abs, Add, Divide, Gain, Math Function, MinMax, Product, Sqrt
- **Logic and Bit Operations:** Logical Operator, Relational Operator
- **Signal Routing:** Switch
- **Sinks:** Outport
- **Sources:** Inport, Constant

Documentation Generation Throughout the development process, documents are used to support the review and further development of the system model, as well as to facilitate communication between the teams involved. The end of function development activities is marked by documenting the approved functions and implementation

of the developed functions to the electronic control unit (ECU) - traditionally accomplished with the use of low-level languages. Furthermore, Simulink's in-built tool Simulink Coder facilitates the automatic generation of C code for embedded processors in mass generation, thus making the use of C code in the input prompt to generate the corresponding documentation apparent. Although Simulink Coder creates multiple files, the finalised prompt input is truncated to contain only the description of the function and the variables to limit token usage while ensuring clarity. A standard automotive output documentation normally consists of a description of the variables in the instruction (inputs, outputs, and constants), their type, and description, along with a short description of the function.

A function development instruction-output instance contains the software requirement for the function as an instruction, and the corresponding output is MATLAB code that generates a Simulink model for the specified requirement. Similarly, an instruction-output instance for documentation generation contains the description of the function written in C, and the corresponding output is a documentation report of the given function. The dataset used as the seed tasks in this study was meticulously constructed using various project documentation examples and active consultation with FEV Europe GmbH. Limited by the availability and suitability of the datasets, and taking into account confidentiality issues, an initial pool of 46 seed tasks was finalized.

2.4. Parameter Tuning

Certain parameters can affect the reliability, creativity, and accuracy of the LM-generated instruction-output instances. Focusing on *temperature* and *top_p*, which control the randomness of the generated results and the sampling probability mass for each token generation respectively, experiments were conducted to evaluate the best settings for the aforementioned use cases. Each experiment entailed the generation of 100 instruction-output pairs, of which 30 were randomly sampled and manually evaluated using the scheme detailed in Section 2.2.

Function Development Keeping the *top_p* constant at a value of 0.5, the *temperature* was varied over a range from 0.65-0.9 as shown in Figure 3. Subsequently, keeping the *temperature* constant at a value of 0.9, the *top_p* was varied from 0.5-0.9. Lower *temperatures* generally resulted in generated instructions that were relatively simple and had a high similarity to the original seed tasks used. Higher *temperatures* and *top_p*, both resulted in a more complicated instruction generation. Higher *top_p*, simultaneously, also resulted in more implementation errors in the generated outputs.

Documentation Generation In the case of documentation generation, the *temperature* was varied over a range of 0.65-0.85 while keeping the *top_p* constant at a value of 0.5 as shown in Figure 4. Similarly, keeping the *temperature* constant at a value of 0.85, the *top_p* varied from 0.5-0.8. Although increasing *temperature* increased the diversity of the generated instructions, the quality of the generated data remained largely unaffected. Increasing *top_p*, however, resulted in a more diverse instruction generation while also increasing the reliability of accurately following the variable naming convention.

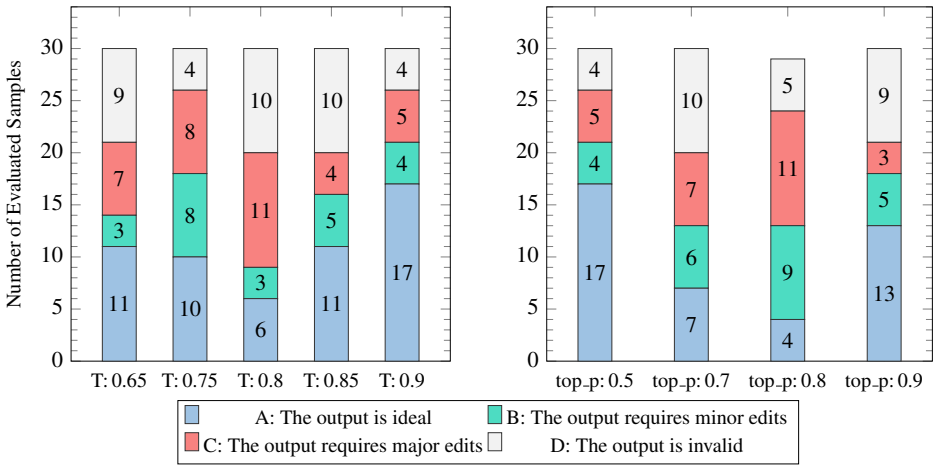


Figure 3: Function development: Human evaluation of Sim-Instruct over various *temperature* and *top_p* ranges

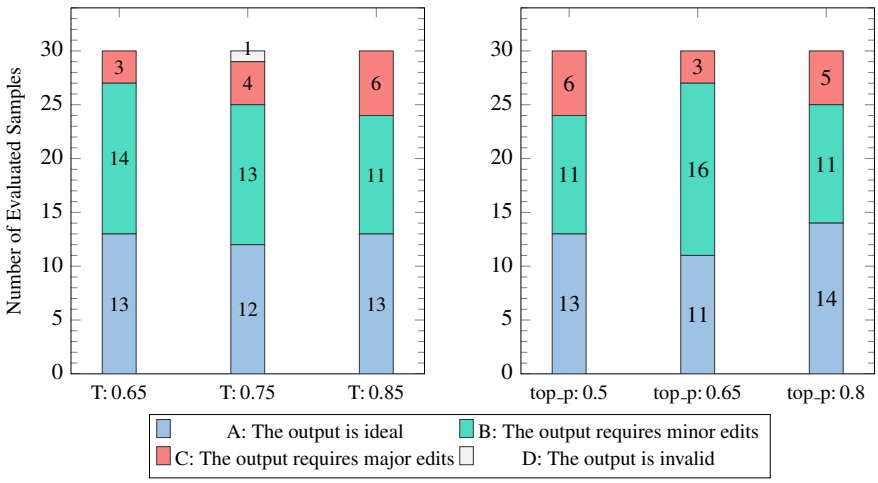


Figure 4: Documentation generation: Human valuation of Sim-Instruct over various *temperature* and *top_p* ranges

The best-determined parameters that maximised reliability, accuracy, and creativity in the generated data for both cases are summarised in Table 1. Throughout this determination process, certain generated instruction-output pairs were added to the initial seed task pool, raising their numbers to 96 seed tasks for function development and 78 seed tasks for documentation generation.

Table 1: Selected parameters for Sim-Instruct

Use Case	Temperature	Top_P
Function Development	0.9	0.5
Documentation Generation	0.85	0.8

2.5. Repairing Algorithm

An initial evaluation of randomly sampled generated instruction-output instances for function development highlighted the LM's limited ability to write correct Simulink code. To ensure the syntactic correctness of the generated Simulink code, an algorithm was developed and implemented to automate the repair of the generated output where possible and to filter out irreparable syntax errors. The implementation of the repair algorithm, depicted in Figure 5, consists of the following steps: 1) description of encountered error in every output instance, 2) identification and grouping of instances with errors that can be repaired without human intervention, 3) repairing and updating the original output instances, and 4) removal of output instances from the training dataset where no repair was possible. Steps 1-3 are repeated until the desired stopping criteria are reached, where the last step is executed.

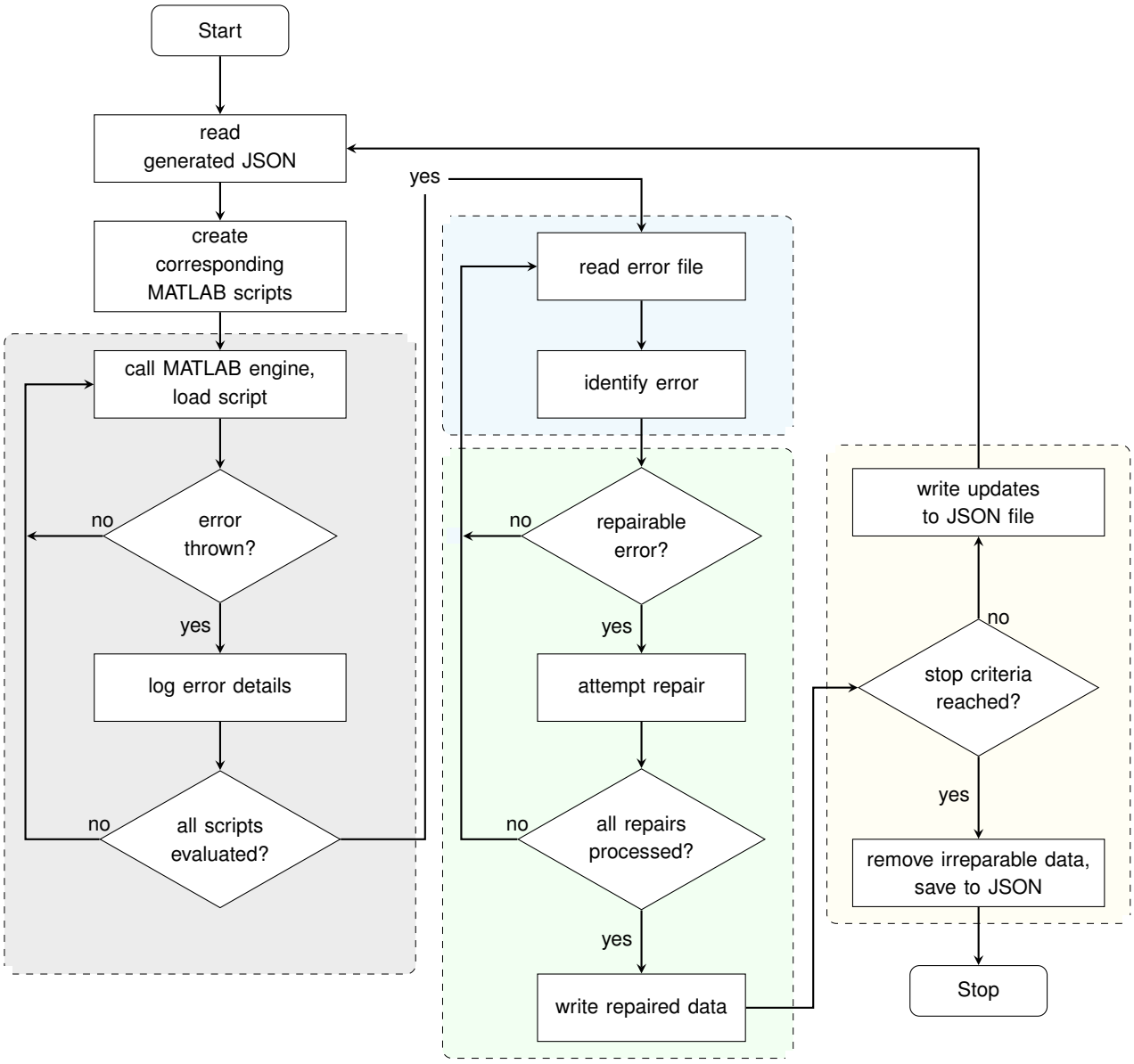


Figure 5: Flow chart summarising the repair algorithm

2.6. *Instruction Fine-Tuning*

The data generated by Sim-Instruct was utilised to fine-tune Code-Llama 7B [9] and Mistral 7B [4]. The prompt, similar to [7], format for fine-tuning is outlined as follows:

> Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.
>
> ### Instruction:
> {instruction}
>
> ### Response:
> {response}

Integrating 4-bit quantization techniques, a parameter efficient fine-tuning with Low-Rank Adaptation (LORA) [30] was adopted. This enabled the accommodation of the models within two Tesla V100 GPUs, each equipped with 16GB of GPU memory. The learning rate for Code-Llama 7B was set to $\eta = 3 \cdot 10^{-4}$ following recommendations from [9], while for Mistral 7B, the learning rate was adjusted to $\eta = 2 \cdot 10^{-4}$. Following the original Llama 2 and Code-Llama work [6, 9] on specialization of LLMs while preventing over-fitting, the number of training epochs was capped at 2. The batch size was set to 16 and gradient accumulation steps was set to 8 to reduce GPU memory requirements.

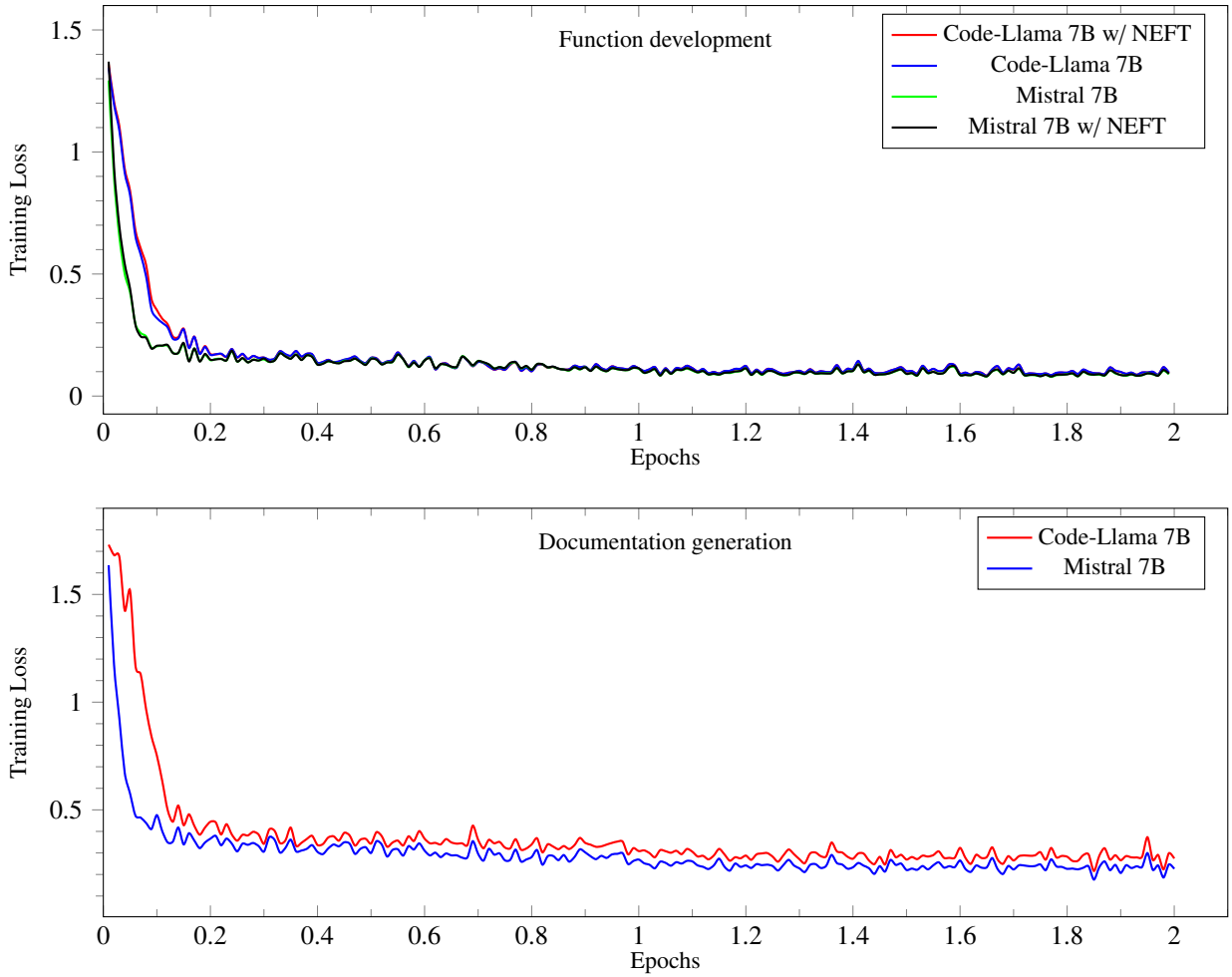


Figure 6: Training loss during instruction tuning.

Additionally, the LLMs were once more trained with the noisy embedding instruction fine-tuning (NEFTune) approach [31], which adds noise to the embedding vectors during training. Figure 6 shows the training loss for the fine-tuning of Code-Llama 7B and Mistral 7B over our training data - the training loss quantifies the discrepancy between the LLM's predicted outputs and the desired outputs. Lower training loss indicates improved alignment between the LLM's predictions and the ground truth, reflecting progress in learning the target task. The downward trend in training loss for both function development and documentation generation indicates that the LLMs are effectively adjusting their parameters to better fit the given data.

3. Results & Discussion

In this section, we begin by evaluating the synthetic data generated by Sim-Instruct, with a focus on assessing its quality and diversity. We then provide a detailed overview of the benchmark we developed, which is designed to test and compare our approach against the current state-of-the-art LLMs. Finally, we evaluate the performance of our fine-tuned models using this benchmark.

3.1. Sim-Instruct

This subsection discusses the data generated using the Sim-Instruct framework. A total of 2500 instruction-output instances were generated as the initial step for both the function development and documentation generation tasks. To assess the quality of the generated data, a random sample of 100 instruction-output instances was evaluated using the scheme described in Section 2.2. The evaluation is summarised in Figure 7. In the case of documentation generation, almost all generated data is meaningful and only 1 of the evaluated instruction-output instances was deemed unusable. However, in the case of function development, the generated data is of lower quality, and almost 25% is deemed unusable. This was expected, as Simulink modelling is inherently more complex than documentation generation due to the specific syntax and function-dependent block connections. The classification of the generated data as D can be attributed to multiple factors: (1) incorrect or incomplete generated requirement leading to an incoherent implementation in the output, (2) inability to implement the generated requirement, and (3) syntactical errors in the generated outputs. The generated instructions were sufficiently diverse with a slight preference towards reuse of variable names appearing in the seed tasks and easier instructions without any if-else conditionals.

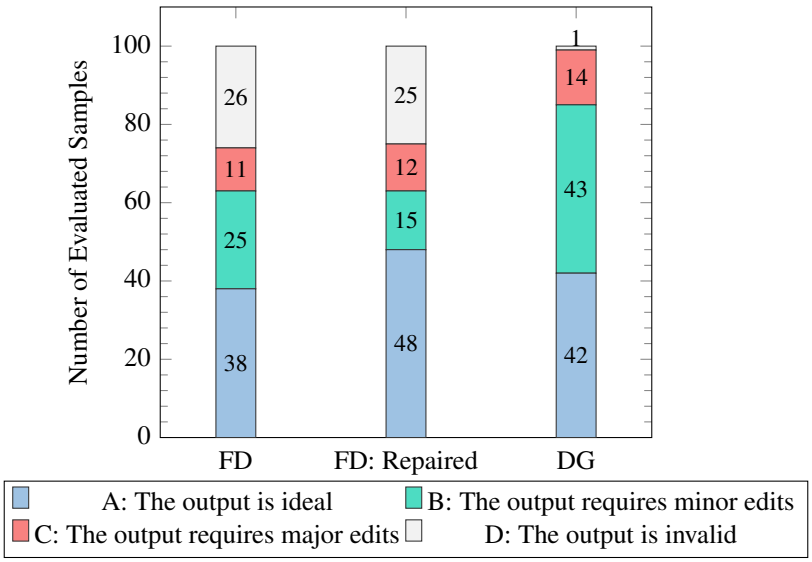


Figure 7: Evaluation of generated instruction-output instances: Function development (FD) and documentation generation (DG).

A complete error analysis of the LLM-generated instruction-output instances for the function development task concluded that 36% of the generated output codes contained syntactical errors. The generated output codes were

evaluated in MATLAB and the thrown error exceptions along with the identifiers were documented. The type and description of the errors found are listed in the following table:

Table 2: Summary of identified errors in the generated instruction-output instances.

Error	**Error Description**
Inability to add a new block	• input and output name overlap • input and constant name overlap • constant and output name overlap • any combination of the above
Invalid parameter setting used	• declared block does not contain declared parameter • wrong parameter setting used
Invalid block specifier	• incorrect library used during declaration • incorrect block name used during declaration
Inability to add a new line to block	• line cannot be added to the destination block
Invalid Simulink object name	• improper or missing object/block definition • attempting to connect more than the permitted lines to an operator • line origin from an output block
Unknown parameter used, undefined function	• declaring a non-existent parameter • any spelling errors in the generated code • arbitrary syntax errors

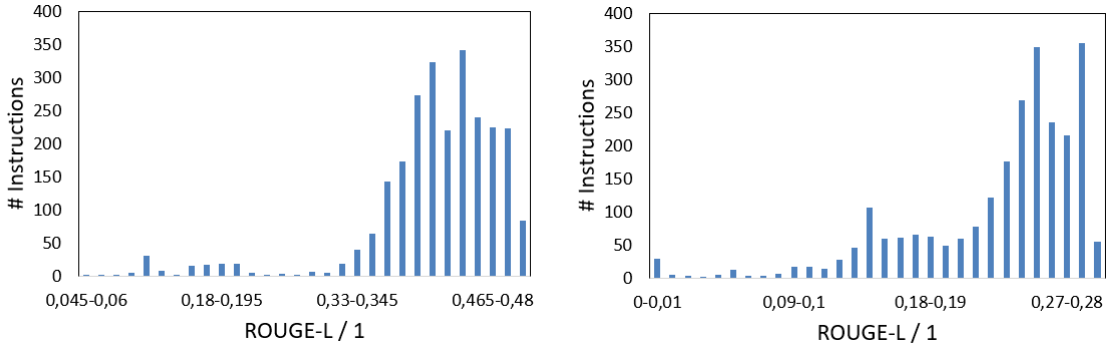
The repair algorithm, described in section 2.5, repaired 53% of the faulty generated training data. 423 generated instruction-output instances were discarded altogether. A random sample of 100 instruction-output instances was subsequently evaluated from the filtered dataset, as shown in Figure 7. The percentage of instances graded as 'A' increased by 10%. However, as the repair algorithm only corrects a portion of syntactical errors, the 25% share of instances evaluated as 'D' stems from the semantic incorrectness of either the generated output or an incoherent instruction as listed above. Table 3 summarises the statistics for the instruction-output generation and the subsequent repair step for both function development and documentation generation tasks.

Table 3: Statistics about the generated instruction-output instances.

Function development	total generated instances	2500
	errors found	986
	errors repaired	563
	errors deleted	423
	instances used for fine-tuning	2077
	mean instruction length (in words)	16
	mean output length (in words)	113
Documentation generation	total generated instances	2500
	instances used for fine-tuning	2500
	mean instruction length (in words)	12
	mean output length (in words)	83

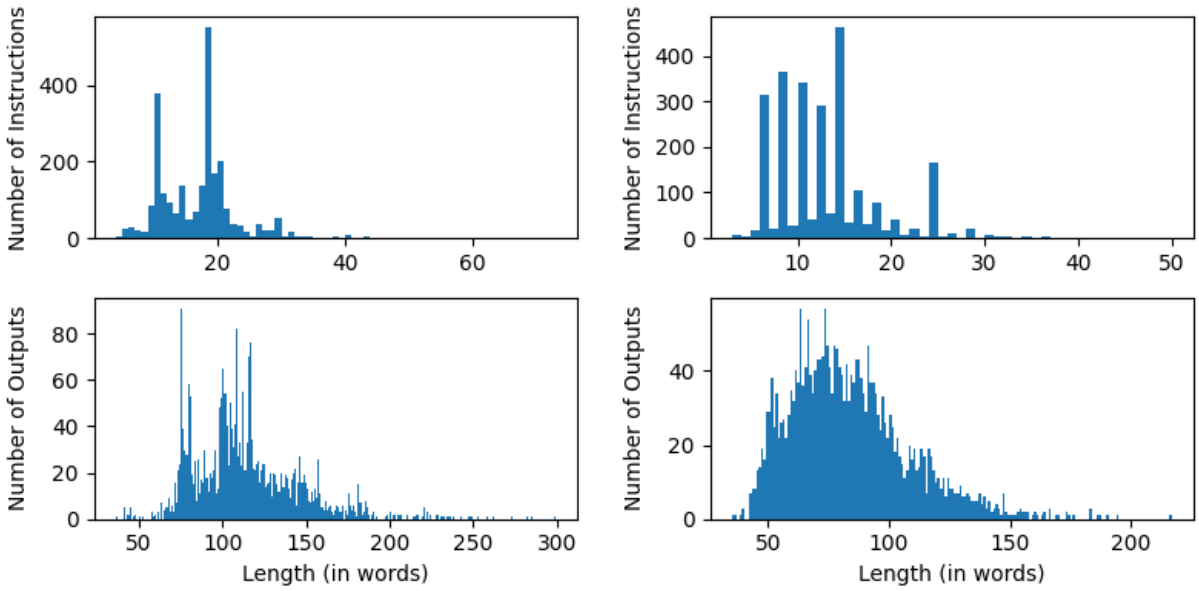
To evaluate the diversity of the generated instruction-output instances, the ROUGE-L metric is used. The ROUGE-L metric measures sentence-to-sentence similarity by measuring the overlapping sub-sequences and long-range de-

pendencies between the generated and reference text [32]. For each newly generated instruction, the highest ROUGE-L overlap with the seed tasks was computed. Given two string sequences, X and Y, the ROUGE-L measure is 1, when X = Y, and the measure is 0 when there is nothing in common between X and Y. The resulting distribution of ROUGE-L scores is presented in Figure 8. The maximum resulting overlap for function generation is under 48% and the maximum overlap for documentation generation is under 30% indicating the generation of mostly dissimilar new instructions. Figure 9, demonstrates the variation in the lengths of instructions and instance outputs displaying a substantial diversity in the generation instructions.



(a) Function development **(b)** Documentation generation

Figure 8: ROUGE-L overlaps with the most similar seed instructions.



(a) Function development **(b)** Documentation generation

Figure 9: Length of generated instruction-output instances.

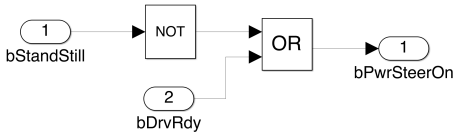
3.2. Benchmarking

In order to assess the effectiveness of the fine-tuned LLMs, a set of 10 instructions was selected as benchmarks for the function development. This was done after consultation and collaboration with experts at FEV Europe GmbH. Among these instructions, 5 were labelled as 'easy' and the remainder as 'medium' difficulty. Specifically, these 10 instructions were curated and categorised based on several factors, such as the number of operations required, the number of blocks needed, and the overall complexity of the instruction. Table 4 presents one example for each group.

Table 4: Examples of easy and medium problems within the benchmark.

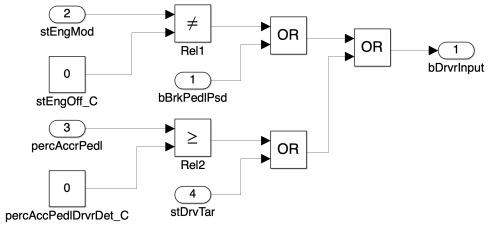
Easy

```
the system shall always satisfy if
((!bStandStill)|(bDrvRdy)) then
(bPwrSteerOn)
```



Medium

```
the system shall always
satisfy if ((bBrkPedlPsd) |
((stEngMod)!=(stEngOff_C)) |
((percAccrPedl)>=(percAccPedlDrvrDet_C))
| (stDrvTar)) then (bDrvrInput)
```



A series of unit tests have been developed to automatically assess the quality of the generated code for these benchmarks. Table 5 illustrates the structure of these tests. Given a generated output code for an instruction, the tests contain the input variable names and pairs of input values and the corresponding expected ideal outputs. In total, 116 tests are designed to cover all edge and rare cases.

Table 5: Examples of prepared unit tests for problems within the benchmark.

Instruction	Tests
`the system shall always satisfy if ((!bStandStill)|(bDrvRdy)) then (bPwrSteerOn)`	input_names: *[bStandStill, bDrvRdy]*, { test_id: *1.1*, input_values: *[0, 0]*, expected_value: *1* } { test_id: *1.3*, input_values: *[10, 0]*, expected_value: *0* } { ... }
`the system shall always satisfy if ((bBrkPedlPsd) | ((stEngMod)!=(stEngOff_C)) | ((percAccrPedl)>=(percAccPedlDrvrDet_C)) | (stDrvTar)) then (bDrvrInput)`	input_names: *[stEngMod, stEngOff_C, bBrkPedlPsd, percAccrPedl, percAccPedlDrvrDet_C, stDrvTar]* { test_id: *6.1*, input_values: *[10,10,0,5,5,0]*, expected_value: *1* } { test_id: *6.2*, input_values: *[10,10,0,5,5,2]*, expected_value: *1* } { ... }

Code generative models are primarily benchmarked against reference solutions, where the match can be exact or fuzzy. However, match-based metrics like BLEU score [33] may not capture code-specific semantic features well [34]. Alternatively, some studies focus on whether the generated code passes unit tests. This line of work aligns with how human developers judge code, following principles like test-driven development. To assess the effectiveness of the generated code in function development tasks, we calculate the percentage of tests that each model passes successfully. For each problem in the benchmark, a single sample is generated, and the resulting code is evaluated against the unit tests. We then report the fraction of tests that the generated code successfully passes.

The outputs of LLMs were tested and compared in three different scenarios: (1) without fine-tuning or examples provided in the prompt (0-shot), (2) without fine-tuning but with 3 examples provided in the prompt (3-shot), and (3) with fine-tuning and no examples provided. Table 6 presents a comparison of different models' outputs based on the percentage of tests passed. As anticipated, employing the LLMs without fine-tuning or providing examples results in extremely poor performance, with none of the 116 tests being passed. This underscores the challenging nature of the problem at hand and highlights the limited availability, if any, of Simulink data during the pre-training phase of these LLMs. In the 3-shot setting, GPT-4 outperforms the other LLMs, passing almost 40% of the unit tests, while Mistral 7B passes the second-highest number of tests at 19.8%. Fine-tuned Mistral 7B and Code-Llama 7B passes 31% and

44.8% of the unit tests respectively. NEFTune, furthermore, significantly improved the performance of Mistral 7B over the benchmark with a pass percentage of 59.5%. However, NEFTune resulted in a worse performance for Code-Llama 7B, with the pass percentage decreasing to 21.5%. The results, thus, suggest that in the function development task fine-tuning outperforms in-context learning.

Table 6: Evaluation of various models for function development based on the percentage of unit tests passed.

Base model	**Fine-tuned**	**NEFT**	**Passed %**
Mistral 7B (0-shot)	×	×	0.0
Mistral 7B (3-shot)	×	×	19.8
Mistral 7B	✓	×	31.0
Mistral 7B	✓	✓	**59.5**
Code-Llama 7B (0-shot)	×	×	0.0
Code-Llama 7B (3-shot)	×	×	12.0
Code-Llama 7B	✓	×	44.8
Code-Llama 7B	✓	✓	21.5
GPT-3.5-turbo (0-shot)	×	×	0.0
GPT-3.5-turbo (3-shot)	×	×	6.9
GPT-4 (0-shot)	×	×	0.0
GPT-4 (3-shot)	×	×	39.5

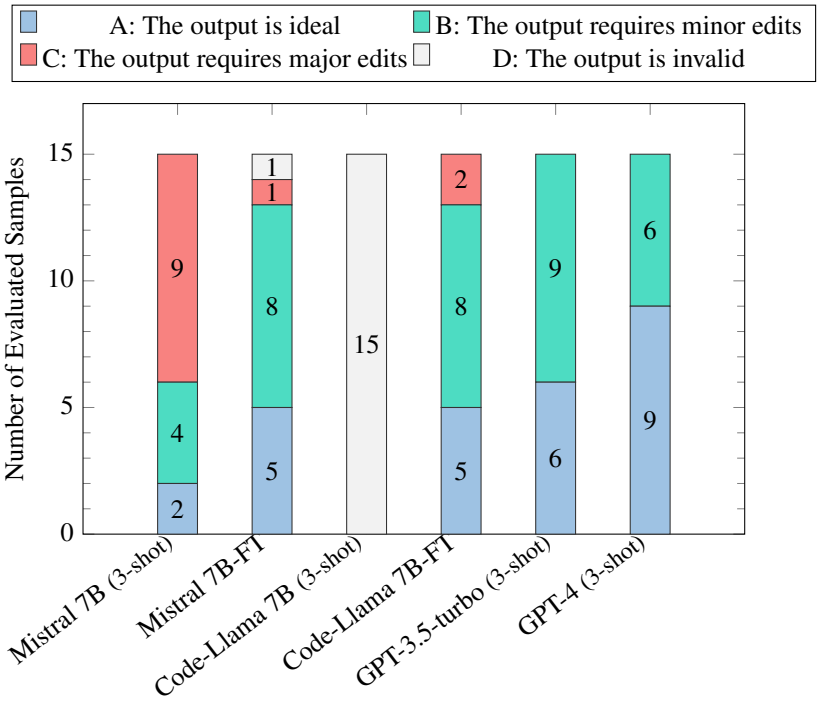


Figure 10: Evaluation of various models for documentation generation.

In the case of documentation generation, the output of various LLMs is evaluated manually against a set of 15 instructions and categorised according to the scheme detailed in Section 2.2. The results are documented in Figure 10. The fine-tuned Mistral-7B and CodeLlama-7B (represented in Figure 10 as Mistral7B-FT and Code-Llama 7B-FT, respectively). The base versions of the two LLMs along with GPT-3.5-turbo and GPT-4 were prompted with 3-shot

to provide the models with examples of the expected documentation template. Figure 10 shows that the fine-tuning did not result in enhanced performance than the GPT-3.5-turbo model which was originally used for the training data generation. Although, fine-tuning improves the performance against few-shot prompting for Mistral and Code-Llama, fine-tuning did not improve the results over 3-shot prompting of GPT-3.5-turbo.

4. Conclusion

In this paper, the goal was to leverage LLMs for function development in graphical programming, specifically in the MATLAB Simulink environment with the aim of bringing the recent advancements in LLM-based code generation to the graphical programming processes. To overcome the scarcity of data for training, a novel Sim-Instruct framework with a repair algorithm was developed, combining elements of Self-Instruct and Evol-Instruct methodologies. The framework allowed for the generation of diverse and complex instruction data while mitigating the limitations inherited from the LLM used for data generation.

In order to demonstrate the superiority of the proposed approach, a benchmark was developed to evaluate the LLM's ability to generate functions using graphical programming based on software requirements. The fine-tuned LLMs, despite being significantly smaller in size, outperform GPT-3.5-turbo and GPT-4 by substantial margins. This demonstrates the efficacy of the approach taken. While fine-tuning for function generation in graphical programming yielded significant improvements over GPT-3.5-turbo - the LLM used for training data generation - a similar enhancement was not observed for documentation tasks. This discrepancy can be attributed to the already proficient performance of established LLMs in documentation and summarisation capabilities.

This paper focused on the training and testing of LLMs on small and medium-sized graphical programming functions. Although the results are promising, it is evident that LLMs require further training to tackle more complex function generation tasks within graphical programming. Future research will incorporate more challenging functions in the training and testing datasets to enhance the utility of the fine-tuned LLMs. Additionally, alternative fine-tuning approaches will be investigated, including reinforcement learning with human feedback (RLHF) and reinforcement learning with AI feedback (RLAIF), or using bigger open-source models instead of relying on rather small models, which can further improve the performance of LLMs in the graphical programming context.

Acknowledgement

Computations were performed with computing resources granted by RWTH Aachen University under project rwth1475.

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.

[2] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam,

K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, B. Zoph, Gpt-4 technical report (2024). arXiv:2303.08774.

[3] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, R. Lowe, Training language models to follow instructions with human feedback, arXiv e-prints (2022). arXiv:2203.02155, doi:10.48550/arXiv.2203.02155.

[4] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed, Mistral 7b (2023). arXiv:2310.06825.

[5] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, G. Lample, Llama: Open and efficient foundation language models (2023). arXiv:2302.13971.

[6] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, T. Scialom, Llama 2: Open foundation and fine-tuned chat models (2023). arXiv:2307.09288.

[7] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, D. Jiang, Wizardcoder: Empowering code large language models with evol-instruct (2023). arXiv:2306.08568.

[8] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, Starcoder: may the source be with you! (2023). arXiv:2305.06161.

[9] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, G. Synnaeve, Code llama: Open foundation models for code (2024). arXiv:2308.12950.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, W. Zaremba, Evaluating large language models trained on code (2021). arXiv:2107.03374.

[11] V. Vikram, C. Lemieux, R. Padhye, Can large language models write good property-based tests? (2023). arXiv:2307.04346.

[12] S. Kang, J. Yoon, S. Yoo, Large language models are few-shot testers: Exploring llm-based general bug reproduction (2023). arXiv:2209.11515.

[13] S. Kang, B. Chen, S. Yoo, J.-G. Lou, Explainable automated debugging via large language model-driven scientific debugging (2023). arXiv:2304.02195.

[14] S. Peng, E. Kalliamvakou, P. Cihon, M. Demirer, The impact of ai on developer productivity: Evidence from github copilot (2023). arXiv:2302.06590.

[15] H. Altinger, F. Wotawa, M. Schurius, Testing methods used in the automotive industry: results from a survey, in: Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing, JAMAICA 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 1–6. doi:10.1145/2631890.2631891.
URL https://doi.org/10.1145/2631890.2631891

[16] A. Rau, On model-based development: A pattern for strong interfaces in simulink, Softwaretechnik-Trends 22 (1) (2002).

[17] A. Boll, F. Brokhausen, T. Amorim, T. Kehrer, A. Vogelsang, Characteristics, potentials, and limitations of open-source simulink projects for empirical research (dec 2021). doi:10.1007/s10270-021-00883-0.
URL https://doi.org/10.1007/s10270-021-00883-0

[18] A. Boll, T. Kehrer, On the replicability of experimental tool evaluations in model-based development - lessons learnt from a systematic literature review focusing on matlab/simulink, in: Ö. Babur, J. Denil, B. Vogel-Heuser (Eds.), Systems Modelling and Management - First International Conference, ICSMM 2020, Bergen, Norway, June 25-26, 2020, Proceedings, Vol. 1262 of Communications in Computer and Information Science, Springer, 2020, pp. 111–130. doi:10.1007/978-3-030-58167-1_9.
URL https://doi.org/10.1007/978-3-030-58167-1_9

[19] S. L. Shrestha, Harnessing large language models for simulink toolchain testing and developing diverse open-source corpora of simulink models for metric and evolution analysis (2023). doi:10.1145/3597926.3605233.
URL https://doi.org/10.1145/3597926.3605233

[20] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, H. Hajishirzi, Self-instruct: Aligning language models with self-generated instructions (2023). arXiv:2212.10560.

[21] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu, G. Wang, Instruction tuning for large language models: A survey (2024). arXiv:2308.10792.

[22] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, D. Jiang, Wizardlm: Empowering large language models to follow complex instructions (2023). arXiv:2304.12244.
[23] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, What is the benefit of a model-based design of embedded software systems in the car industry?, IGI Global, 2011. doi:10.4018/978-1-61350-438-3.ch013.
[24] T. Gerlitz, Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts, Dissertation, RWTH Aachen University, Aachen, veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Dissertation, RWTH Aachen University, 2017 (2017). doi:10.18154/RWTH-2017-08881.
URL https://publications.rwth-aachen.de/record/706814
[25] D. Inkermann, T. Huth, T. Vietor, A. Grewe, C. Knieke, A. Rausch, Model-based requirement engineering to support development of complex systems, Procedia CIRP 84 (2019) 239–244, 29th CIRP Design Conference 2019, 08-10 May 2019, Póvoa de Varzim, Portgal. doi:https://doi.org/10.1016/j.procir.2019.04.345.
URL https://www.sciencedirect.com/science/article/pii/S2212827119309990
[26] G. Liebel, N. Marko, M. Tichy, A. Leitner, J. Hansson, Assessing the state-of-practice of model-based engineering in the embedded systems domain, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran (Eds.), Model-Driven Engineering Languages and Systems, Springer International Publishing, Cham, 2014, pp. 166–182.
[27] K. Pohl, Requirements Engineering: Fundamentals, Principles, and Techniques, 1st Edition, Springer Publishing Company, Incorporated, 2010.
[28] A. Mavin, P. Wilkinson, A. Harwood, M. Novak, Easy approach to requirements syntax (ears), in: 2009 17th IEEE International Requirements Engineering Conference, 2009, pp. 317–322. doi:10.1109/RE.2009.9.
[29] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Schumann, Automated formalization of structured natural language requirements, Information and Software Technology 137 (2021) 106590. doi:https://doi.org/10.1016/j.infsof.2021.106590.
URL https://www.sciencedirect.com/science/article/pii/S0950584921000707
[30] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, W. Chen, Lora: Low-rank adaptation of large language models, CoRR abs/2106.09685 (2021). arXiv:2106.09685.
URL https://arxiv.org/abs/2106.09685
[31] N. Jain, P. yeh Chiang, Y. Wen, J. Kirchenbauer, H.-M. Chu, G. Somepalli, B. R. Bartoldson, B. Kailkhura, A. Schwarzschild, A. Saha, M. Goldblum, J. Geiping, T. Goldstein, Neftune: Noisy embeddings improve instruction finetuning (2023). arXiv:2310.05914.
[32] C.-Y. Lin, F. J. Och, Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics, in: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04), Barcelona, Spain, 2004, pp. 605–612. doi:10.3115/1218955.1219032.
URL https://aclanthology.org/P04-1077
[33] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.
[34] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, S. Ma, Codebleu: a method for automatic evaluation of code synthesis, arXiv preprint arXiv:2009.10297 (2020).